The Seasonality of Programming

August 2025

6 min read • 1,425 words

Themes: Technology Programming Spiritual

Every few years, the programming world discovers fire again.

Ruby on Rails would make web development effortless. Node.js would unify frontend and backend. React would make UIs predictable. Docker would solve deployment. Kubernetes would orchestrate everything. Now it's AI that will write our code for us.

Each time, we get excited about the new framework that will finally solve state management. The database that will revolutionize how we store data. The architecture pattern that will make our code maintainable forever.

Then, quietly, we discover the same problems that plagued the last solution. Performance issues. Complexity creep. Vendor lock-in. The learning curve that seemed manageable becomes a mountain. The community fragments. Enthusiasm wanes.

By then, the next fire has been discovered.

The Eternal Return of Solutions

Most problems we're solving have been solved before. Enterprise software has been handling authentication, data persistence, and distributed systems since before many of us were born.

The solutions weren't glamorous. Java, C#, COBOL. WebSphere. Built by people in suits who went home at 5 PM and didn't tweet about their code.

But it worked. Billions of transactions, millions of users, years without outages. The techniques—connection pooling, caching, load balancing—weren't revolutionary. Just competent engineering applied consistently.

Why We Keep Reinventing

The seasonality of programming isn't accidental. It serves psychological and economic needs that have little to do with technical necessity.

Novelty bias makes new solutions seem better regardless of merit. The psychological reward of learning something new beats the mundane satisfaction of mastering something old

> The tech industry's addiction to novelty mirrors social media's dopamine feedback loops—both optimize for engagement over depth.

.

Career anxiety drives adoption. Résumés with React and Kubernetes get callbacks. jQuery and monoliths don't, even when they'd work better.

Venture capital needs to justify valuations by convincing developers that existing solutions are fundamentally inadequate.

Conference economics require fresh content. You can't give the same talk about database normalization for ten years, even if it's still important.

None of this is inherently malicious. But it creates systemic pressure to treat proven solutions as obsolete and experimental technologies as mature.

The Hidden Costs of Seasonal Programming

What looks like innovation from inside the tech industry often looks like chaos from outside it. While we chase the latest architectural patterns and deployment strategies, we lose sight of the human costs of our optimization choices.

Technical Debt as Social Debt

Every rewrite, every migration to the new framework, every adoption of the revolutionary database creates work that doesn't directly serve users. The months spent moving from Angular to React to Vue could have been spent improving accessibility. Fixing performance issues. Adding features users actually needed.

This technical churn becomes social debt when products become less reliable, more complex, or harder to maintain because teams are constantly rebuilding foundations instead of improving the house. The instability we create in our codebases creates instability in the products people depend on.

Complexity as Barrier

Each new layer of sophistication raises the barrier for participation. The simple LAMP stack that enabled millions to build their first websites? Gone. Replaced by containerized microservices requiring specialist knowledge.

This complexity doesn't just affect developers—it excludes businesses, nonprofits, and communities that can't afford cutting-edge engineering talent

> The same complexity that excites Silicon Valley engineers makes technology less accessible to those who need it most.

.

The Optimization Mismatch

Most critically, we optimize for metrics that matter to us—developer experience, deployment velocity, architectural purity—while remaining largely unaware of how these choices affect the people who use our software.

The algorithm eats virtue because we optimize for engagement over well-being. It consumes democratic discourse because we optimize for growth over social cohesion. It fragments reality because we optimize for personalization over shared understanding.

The seasonal nature of programming amplifies this mismatch. By constantly chasing new tools and techniques, we avoid reckoning with the consequences of how we've been building software. The next framework will fix everything, so we don't need to examine whether our current approach serves human flourishing.

The Unix Philosophy at Scale

Ted Dziuba made this point brilliantly in his essay on "Taco Bell Programming": most complex problems can be solved by combining a small set of well-understood tools in clever ways, just like how Taco Bell builds their entire menu from six basic ingredients.

The same principle works for programming. Need to transform data across millions of files? The trendy answer involves Apache Spark clusters and distributed computing frameworks. The simple answer? `awk`, `sed`, and `parallel`. Need to monitor system performance? Skip the observability platform — `top`, `iostat`, and `grep` through logs will tell you what's actually wrong. Need to sync files across servers? Before you deploy that distributed filesystem, try `rsync` and `cron`.

Every new service or framework you introduce creates potential failure points. Basic Unix tools have been battle-tested for decades. They fail in predictable ways. They have manuals. They don't require learning new APIs or fighting with dependency management. Most importantly, they don't wake you up at 3 AM with mysterious errors.

The Enterprise Wisdom

Enterprise software gets mocked for being boring, slow to change, risk-averse. But enterprise engineers often have constraints that force better long-term thinking: they maintain systems for decades, support thousands of users, meet regulatory requirements, work with limited budgets.

These constraints create different priorities. Enterprise developers care more about reliability than novelty, maintainability than elegance, gradual evolution than revolutionary change. They use technologies that are proven, documented, supported by vendors who will be around in ten years.

The results aren't always pretty by startup standards. But they work better for people who depend on them. Enterprise software fails less dramatically, scales more predictably, integrates more reliably with existing systems. It prioritizes boring reliability over exciting innovation.

Sustainable Programming Seasons

The seasonality of programming isn't inherently harmful—technology does advance, new tools do solve real problems, and intellectual curiosity drives important innovation. The issue is when seasonal change becomes compulsive change, when novelty becomes the primary value.

Sustainable programming seasons would look different:

- **Longer cycles.** Instead of adopting new frameworks every year, we might evaluate them every three to five years. Give time to understand their real-world performance and community stability.
- **Problem-first adoption.** New technologies would be evaluated based on specific problems they solve rather than their general appeal or industry buzz.
- **Total cost accounting.** Technology decisions would consider not just development velocity but maintenance burden, security implications, talent availability, and migration costs.
- **User impact assessment.** We'd ask how technology choices affect the people who use our software, not just the people who build it.
- **Institutional memory.** Teams would maintain knowledge of why previous technologies were chosen and what problems they solved, avoiding the cycle of rediscovering old solutions.

Beyond the Hype Cycle

The most important software comes from people who step outside the cycle. They use whatever solves the problem. They focus on user needs, not industry trends.

Early adoption can be valuable when it serves genuine innovation. But most seasonal programming is just middle adoption—following trends, not setting them.

The alternative isn't conservatism. It's intentionality. Choose tools based on problems, not popularity. Measure success by impact, not adoption metrics.

The Responsibility of Seasons

Programming seasonality will continue as long as human psychology remains unchanged. We'll always be drawn to new solutions, excited by fresh approaches, motivated by the promise that this time will be different.

The question is whether we can develop awareness of these patterns and their consequences. Whether we can balance our appetite for innovation with responsibility for the systems we build and maintain.

Whether we can remember that software is ultimately a tool for serving human needs, not satisfying technical curiosity.

The seasons will change. The question is what we choose to plant, and what we're willing to harvest.

This examination of cyclical technology adoption builds from early insights in Early Adoption, which explores pattern recognition and distinguishes genuine innovation from trend-following. The human costs of optimization choices unfold in The Algorithm Eats Virtue, where engagement optimization systematically undermines flourishing, while the fragmentation of collective decision-making emerges in The Algorithm Eats Democracy. The alternative approach of conscious technology development appears in Programming as Spiritual Practice, and the broader analysis of optimization's impact on human values continues in Algorithmic Critique.

These patterns find their business analysis in Clayton Christensen's examination of why successful companies struggle with disruption in The Innovator's Dilemma, connect to Geoffrey Moore's investigation of adoption cycles in Crossing the Chasm, ground in Don Norman's user-centered design principles that transcend technological trends in The Design of Everyday Things, and extend through Nadia Eghbal's exploration of open source community dynamics in Working in Public.
